

Plexon Inc

Reading PLX and DDT Files with Matlab

Issued [11/22/2005] – Plexon Inc

CONFIDENTIAL

© Copyright [2005-2013] Plexon Inc

® Plexon is a Registered trademark of Plexon Inc

**Plexon Inc
6500 Greenville Ave. LB33
Dallas TX 75206**

**Tel: 214 369 4957
Fax: 214 369 1775
www.plexon.com**

Reading PLX and DDT Files with C/C++

This package provides the information required to read PLX and DDT files. A PLX file is a Plexon data file containing action-potential (spike) timestamps and waveforms (spike channels), event timestamps (event channels), and continuous variable data (continuous A/D channels). A DDT file is a Plexon continuous data file optimized for continuous (streaming) recording where every channel is continuously recorded without gaps and the recording includes any “dead time” between spikes.

This package contains the following:

- Reading PLX and DDT Files with Matlab.doc – this document
- Plexon.h – header file that describes the structure of a PLX and DDT file
- Sample Win32 console application for reading PLX files - source code is located in the plxReader_Win32_Console folder
- Sample MFC application for reading PLX files – source code located in the plxReader_MFC folder
- Sample Win32 console application for reading DDT files - source code is located in the ddtReader_Win32_Console folder
- Sample MFC application for reading DDT files – source code located in the ddtReader_MFC folder
- Sample Win32 console application for decoding CinePlex tracking coordinates from a PLX file - source code is located in the ddtReader_Win32_Console folder
- Sample MFC application for decoding CinePlex tracking coordinates from a PLX file – source code located in the ddtReader_MFC folder
- Sample PLX file (test1.plx) – located in the SampleData folder
- Sample DDT file (test1.ddt) – located in the SampleData folder

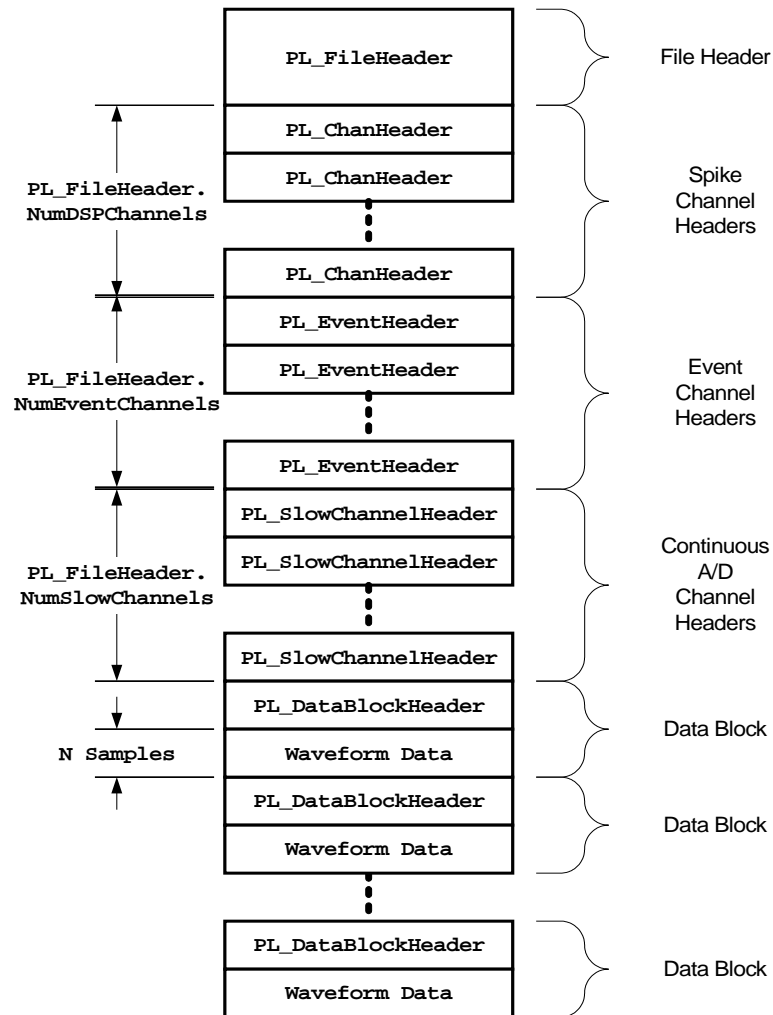
If you have any questions about reading PLX and/or DDT files, please feel free to contact Plexon at support@plexon.com

Contents

1	PLX File Structure	4
1.1	File Header	5
1.2	Spike Channel Header	6
1.3	Event Channel Header	6
1.4	Continuous Channel Header	6
1.5	Data Block Header.....	7
1.5.1	Spike Data Blocks	8
1.5.2	Event Data Blocks.....	9
1.5.3	Continuous A/D Data Blocks.....	9
2	PLX File Reader Sample Programs	12
2.1	MFC Sample.....	12
2.2	Win32 Console Sample	14
3	DDT File Structure	15
3.1	Version 100.....	16
3.2	Version 101.....	16
3.3	Version 102.....	17
3.4	Version 103.....	17
4	DDT File Reader Sample Programs	19
4.1	MFC Sample Application.....	19
4.2	Win32 Console Sample	20
5	CinePlex Tracking Coordinate Decoding	21

1 PLX File Structure

A PLX file consists of a file header, channel headers, and data blocks as shown in the figure below.



1.1 File Header

The file header specifies general information about the PLX file including the time/date the file was created, global sampling parameters, the number of spike, event, and continuous channels, and a tally of timestamp and waveform counts for each channel. The file header is defined by the PL_FileHeader structure (see header file Plexon.h) :

```
struct PL_FileHeader
{
    unsigned int MagicNumber; // = 0x58454c50;

    int Version; // Version of the data format; determines which data
                // items are valid
    char Comment[128]; // User-supplied comment
    int ADFrequency; // Timestamp frequency in hertz
    int NumDSPChannels; // Number of DSP channel headers in the file
    int NumEventChannels; // Number of Event channel headers in the file
    int NumSlowChannels; // Number of A/D channel headers in the file
    int NumPointsWave; // Number of data points in waveform
    int NumPointsPreThr; // Number of data points before crossing the threshold

    int Year; // Time/date when the data was acquired
    int Month;
    int Day;
    int Hour;
    int Minute;
    int Second;

    int FastRead; // reserved
    int WaveformFreq; // waveform sampling rate; ADFrequency above is
                    // timestamp freq
    double LastTimestamp; // duration of the experimental session, in ticks

    // The following 6 items are only valid if Version >= 103
    char Trodalness; // 1 for single, 2 for stereotrode, 4 for tetrode
    char DataTrodalness; // trodalness of the data representation
    char BitsPerSpikeSample; // ADC resolution for spike waveforms in bits
                          // (usually 12)
    char BitsPerSlowSample; // ADC resolution for slow-channel data in bits
                          // (usually 12)
    unsigned short SpikeMaxMagnitudeMV; // the zero-to-peak voltage in mV for
                                      // spike waveform
                                      // adc values (usually 3000)
    unsigned short SlowMaxMagnitudeMV; // the zero-to-peak voltage in mV for
                                      // slow-channel waveform adc values
                                      // (usually 5000)

    // Only valid if Version >= 105
    unsigned short SpikePreAmpGain; // usually either 1000 or 500

    char Padding[46]; // so that this part of the header is 256 bytes

    // Counters for the number of timestamps and waveforms in each channel and unit.
    // Note that these only record the counts for the first 4 units in each channel.
    // channel numbers are 1-based - array entry at [0] is unused
    int TSCounts[130][5]; // number of timestamps[channel][unit]
    int WFCounts[130][5]; // number of waveforms[channel][unit]

    // Starting at index 300, this array also records the number of samples for the
    // continuous channels. Note that since EVCounts has only 512 entries, continuous
    // channels above channel 211 do not have sample counts.
    int EVCounts[512]; // number of timestamps[event_number]
};
```

1.2 Spike Channel Header

The spike channel header provides general information about the spike channel including its name, channel number, gains/filters, and sorting methods. There is one spike channel header for each spike channel as specified by the NumDSPChannels field of the PL_FileHeader. The spike channel header is defined by the PL_ChanelHeader structure (see header file Plexon.h) :

```
struct PL_ChanelHeader
{
    char    Name[32];           // Name given to the DSP channel
    char    SIGName[32];       // Name given to the corresponding SIG channel
    int     Channel;           // DSP channel number, 1-based
    int     WFRate;            // When MAP is doing waveform rate limiting, this is
                                // limit w/f per sec divided by 10
    int     SIG;               // SIG channel associated with this DSP channel 1 - based
    int     Ref;               // SIG channel used as a Reference signal, 1- based
    int     Gain;              // actual gain divided by SpikePreAmpGain. For pre version
                                // 105, actual gain divided by 1000.
    int     Filter;            // 0 or 1
    int     Threshold;         // Threshold for spike detection in a/d values;
    int     Method;            // Method used for sorting units, 1 - boxes, 2 - templates
    int     NUnits;            // number of sorted units
    short   Template[5][64];   // Templates used for template sorting, in a/d values
    int     Fit[5];            // Template fit
    int     SortWidth;         // how many points to use in template sorting
                                // (template only)
    short   Boxes[5][2][4];    // the boxes used in boxes sorting
    int     SortBeg;           // beginning of the sorting window to use in
                                // template sorting (width defined by SortWidth)

    char    Comment[128];
    int     Padding[11];
};
```

1.3 Event Channel Header

The event channel header provides information about an event channel including its name and channel number. There is one event channel header for each event channel as specified by the NumEventChannels field of the PL_FileHeader. The event channel header is defined by the PL_EventHeader (see header file Plexon.h) :

```
struct PL_EventHeader
{
    char    Name[32];           // name given to this event
    int     Channel;           // event number, 1-based
    char    Comment[128];
    int     Padding[33];
};
```

1.4 Continuous Channel Header

The Continuous A/D channel header provides information about the continuous A/D channel including its name, channel number, sampling frequency, and gains. The continuous channel header is defined by the PL_SlowChannelHeader (see header file Plexon.h) :

```
struct PL_SlowChannelHeader
{
    char    Name[32];           // name given to this channel
    int     Channel;           // channel number, 0-based
    int     ADFreq;            // digitization frequency
    int     Gain;              // gain at the adc card
};
```

```

int    Enabled;           // whether this channel is enabled for taking data, 0 or 1
int    PreAmpGain;        // gain at the preamp

// As of Version 104, this indicates the spike channel (PL_ChainHeader.Channel) of
// a spike channel corresponding to this continuous data channel.
// <=0 means no associated spike channel.
int    SpikeChannel;

char    Comment[128];
int    Padding[28];
};

```

1.5 Data Block Header

Each data block begins with a data block header and may be followed with waveform data. The data block header provides information about the data block including its type, timestamp, channel/unit number, and the number of samples in the waveform data if present. The data block header is defined by the PL_DataBlockHeader structure (see header file Plexon.h) :

```

// The header for the data record used in the datafile (*.plx)
// This is followed by NumberOfWaveforms*NumberOfWordsInWaveform
// short integers that represent the waveform(s)

struct PL_DataBlockHeader
{
    short    Type;                // Data type; 1=spike, 4=Event, 5=continuous
    unsigned short    UpperByteOf5ByteTimestamp; // Upper 8 bits of the 40 bit timestamp
    unsigned long    Timestamp; // Lower 32 bits of the 40 bit timestamp
    short    Channel;            // Channel number
    short    Unit;               // Sorted unit number; 0=unsorted
    short    NumberOfWaveforms;  // Number of waveforms in the data to
                                // follow, usually 0 or 1
    short    NumberOfWordsInWaveform; // Number of samples per waveform in the
                                // data to follow
}; // 16 bytes

```

Every data block has a 5-byte timestamp that represent elapsed time in ticks (sampling periods). The ADFrequency field in the PL_FileHeader structure determines the number of ticks per second. For example, if the ADFrequency is 40,000, then a timestamp representing 1 second would be equal to 40000. 5-byte timestamps require special handling in C/C++. The lower 4-bytes of the time stamp (TimeStamp) and the upper byte of the time stamp (UpperByteOf5ByteTimeStamp) can be packed into a 64-bit LONGLONG data type as below:

```

LONGLONG ts = ((static_cast<LONGLONG>(dataBlock.UpperByteOf5ByteTimeStamp)<<32)
+ static_cast<LONGLONG>(dataBlock.TimeStamp)) ;

```

The following C/C++ code will convert the LONGLONG timestamp from ticks to seconds:

```

double seconds = (double) ts / (double) fileHeader.ADFrequency ;

```

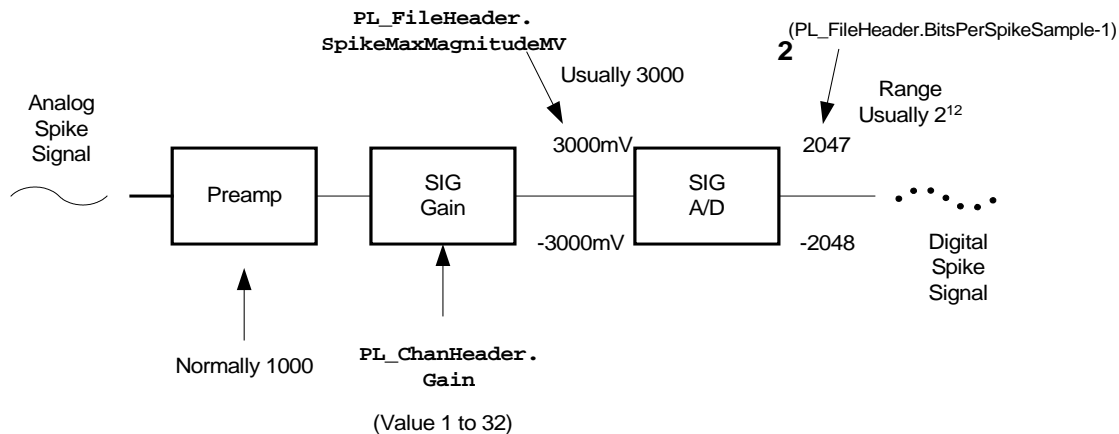
The Type field in the data block header determines whether the data block is a spike data block (PL_SingleWFTType), event data block (PL_ExtEventType), or a continuous A/D data block (PL_ADDataType).

1.5.1 Spike Data Blocks

Spike data blocks (PL_SingleWFTType) correspond to DSP channel headers and record action-potential (spike) timestamps and thresholded waveform segments. The header fields for a spike data block are defined as:

- **TimeStamp** - spike timestamp, time of threshold crossing
- **Channel** - 1-based DSP channel number (1 to number of DSP channels)
- **Unit** - unit number, (0 to 4), 0 = unsorted unit, 1,2,3,4 = sorted units a,b,c,d
- **NumberOfWaveforms** - if 0, there is no waveform for this spike and if 1, spike waveform follows this data block header
- **NumberOfWordsInWaveform** - number of raw A/D values (short integers) in the waveform

If the NumberOfWaveforms field is set to 1, then the data block header is followed by N samples where N is specified by the NumberOfWordsInWaveform field. These samples represent a digitized version of the analog spike signal as shown below:



The preamp is typically configured to provide a gain of 1000 and the SIG gain is specified by the PL_ChainHeader.Gain field which ranges from 1 to 32 in steps of 1. The following equations convert the digital spike samples of the waveform back to the original analog voltage (mV):

For file version < 103

$$\text{Voltage} = \frac{(\text{sample_value})(3000)}{(2048)(\text{PL_ChanHeader.Gain})(1000)}$$

For file version >= 103

$$\text{Voltage} = \frac{(\text{sample_value})(\text{PL_FileHeader.SpikeMaxMagnitudeMV})}{\frac{1}{2}(2^{\text{PL_FileHeader.BitsPerSpikeSample}})(\text{PL_ChanHeader.Gain})(1000)}$$

For file version >= 105

$$\text{Voltage} = \frac{(sample_value)(PL_FileHeader.SpikeMaxMagnitudeMV)}{\frac{1}{2}(2^{PL_FileHeader.BitsPerSpikeSample})(PL_ChanHeader.Gain)(PL_FileHeader.SpikePreAmpGain)}$$

For example, suppose one of the 16-bit digital samples (*sample_value*) is equal to 1000, the A/D reference voltage (*PL_FileHeader.SpikeMaxMagnitudeMV*) is 3000 mV, the bits per sample (*PL_FileHeader.BitsPerSpikeSample*) is 12, the pre-amp gain (*PL_FileHeader.PreAmpGain* for Version >= 105) is 1000, and programmable gain (*PL_ChanHeader.Gain*) for the channel is 2, then the corresponding analog spike voltage is:

$$\text{Voltage} = 1000 * 3000 / (2048 * 2 * 1000) = 732.4 \mu\text{V}$$

1.5.2 Event Data Blocks

Event data blocks (*PL_ExtEventType*) correspond to Event channel headers and record event timestamps. The header fields for event data blocks are defined as:

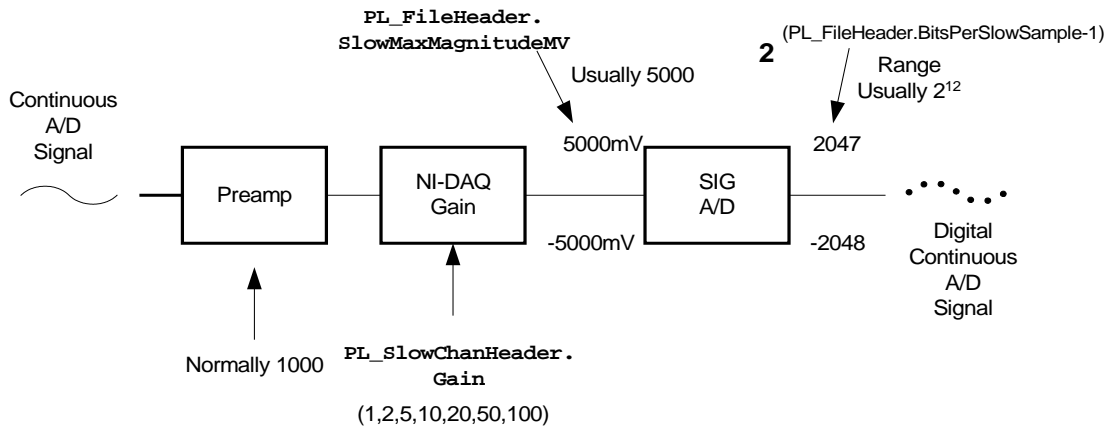
- **TimeStamp** - event timestamp
- **Channel** - 1-based external event channel number (1 to 300)
- **Unit** - if Channel = *PL_StrobedExtChannel* (257), Unit is the strobed value otherwise, Unit is 0

1.5.3 Continuous A/D Data Blocks

Continuous A/D data blocks (*PL_ADDataType*) correspond to continuous A/D channel headers and record raw A/D samples without gaps whereas *PL_SingleWFTType* spike waveforms are thresholded segments of a continuous signal. The header fields for a continuous A/D data block are defined as:

- **TimeStamp** - timestamp of the first A/D value that follow this data block header
- **Channel** - 0-based A/D channel number (0 to 255)
- **Unit** - always 0
- **NumberOfWaveforms** - always 1
- **NumberOfWordsInWaveform** - number of raw A/D values (short integers) that follow this data block header

If the *NumberOfWaveforms* field is set to 1, then the data block header is followed by N samples where N is specified by the *NumberOfWordsInWaveform* field. These samples represent a portion of the digitized version of the analog signal as shown below:



The preamp is typically configured to provide a gain of 1000 and the NI-DAQ gain is specified by the PL_SlowChannelHeader.Gain field having values of 1, 2, 5, 10, 20, 50, and 100. The following equations convert the digital continuous A/D samples of the waveform back to the original analog voltage (mV):

For file versions 100 and 101:

$$\text{Voltage} = \frac{(\text{sample_value})(5000)}{(2048)(\text{PL_SlowChannelHeader.Gain})(1000)}$$

For file version 102:

$$\text{Voltage} = \frac{(\text{sample_value})(5000)}{(2048)(\text{Gain})(\text{PreAmpGain})}$$

where:

- *Gain* is PL_SlowChannelHeader.Gain
- *PreAmpGain* is PL_SlowChannelHeader.PreAmpGain

For file version 103 and greater:

$$\text{Voltage} = \frac{(\text{sample_value})(\text{PL_FileHeader.SlowMaxMagnitudeMV})}{\frac{1}{2}(2^{\text{BitsPerSlowSample}})(\text{Gain})(\text{PreAmpGain})}$$

where:

- *BitsPerSlowSample* is PL_FileHeader.BitsPerSlowSample
- *Gain* is PL_SlowChannelHeader.Gain
- *PreAmpGain* is PL_SlowChannelHeader.PreAmpGain

For example, suppose one of the digital samples (*sample_value*) is equal to 1000, the A/D reference voltage (PL_FileHeader.SlowMaxMagnitudeMV) is 5000 mV, the bits per sample

(`PL_FileHeader.BitsPerSlowSample`) is 12, and programmable gain (`PL_SlowChanHeader.Gain`) for the channel is 2, then the corresponding analog voltage is:

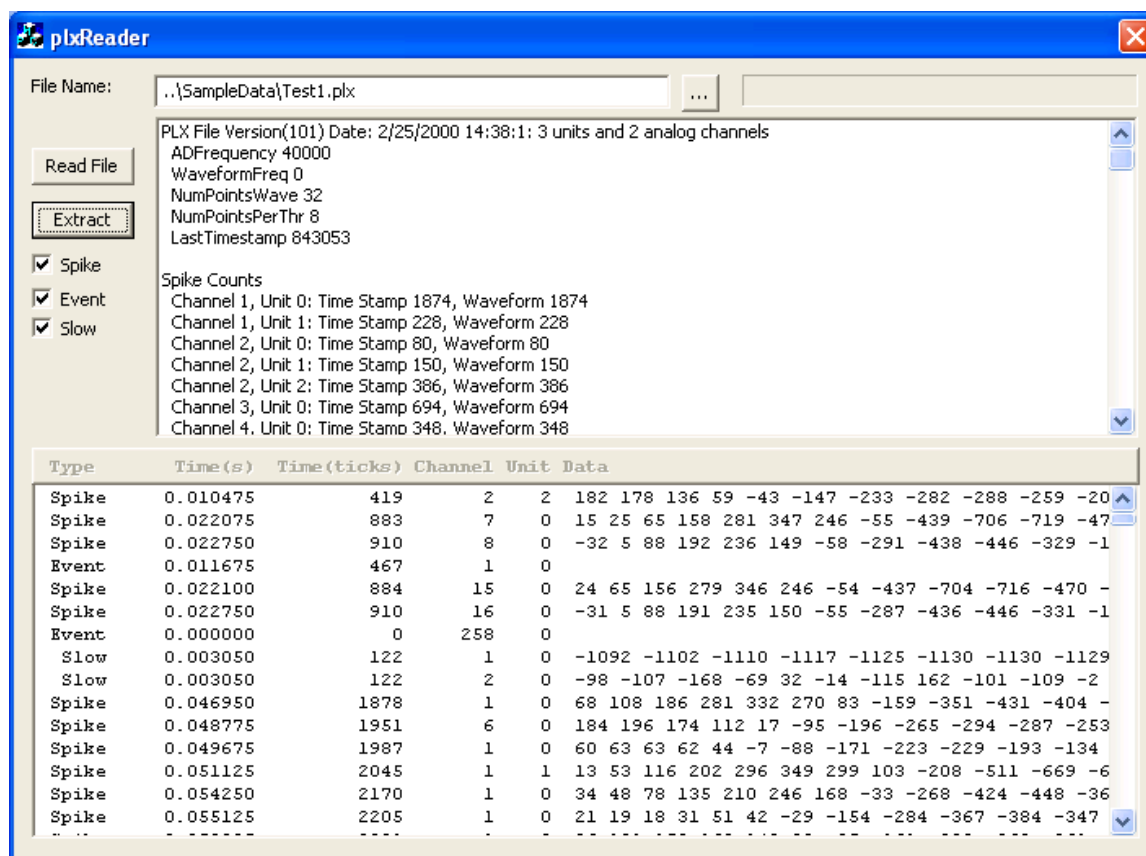
$$\text{Voltage} = 1000 * 5000 / (2048 * 2 * 1000) = 1.2207 \text{ mV}$$

2 PLX File Reader Sample Programs

Two sample applications are provided to illustrate how to read PLX files: an MFC sample and a Win32 console sample. Source code is provided for both samples.

2.1 MFC Sample

The MFC sample application is a dialog based application that displays the contents of a PLX file's header and data as shown below:



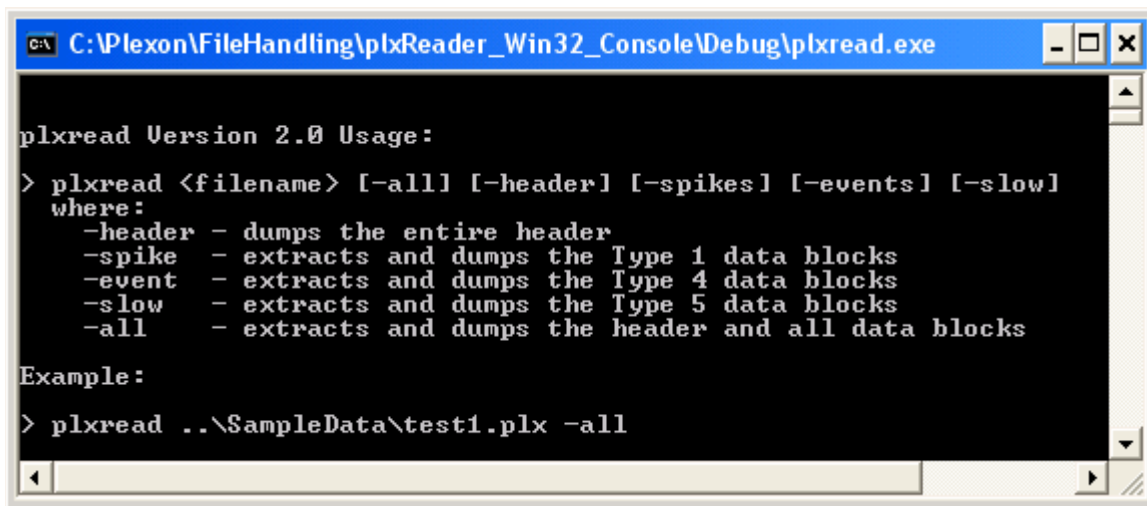
The main files associated with this project are listed in the table below:

File	Description
plxReader.dsp	Microsoft Visual Studio C++ 6.0 project file
plxReaderDlg.h plxReaderDlg.cpp	Contain the header and implementation for the CPlxReaderDlg class. This class implements the main dialog for reading PLX files.
plxReader.h plxReader.cpp	Contains header and implementation for the CPlxReaderApp class. This class contains mostly boilerplate code for the

	main application.
--	-------------------

2.2 Win32 Console Sample

The Win32 console sample application displays the contents of a PLX file's header and data as shown below:



```
C:\Plexon\FileHandling\plxReader_Win32_Console\Debug\plxread.exe

plxread Version 2.0 Usage:
> plxread <filename> [-all] [-header] [-spikes] [-events] [-slow]
where:
  -header - dumps the entire header
  -spike  - extracts and dumps the Type 1 data blocks
  -event  - extracts and dumps the Type 4 data blocks
  -slow   - extracts and dumps the Type 5 data blocks
  -all    - extracts and dumps the header and all data blocks

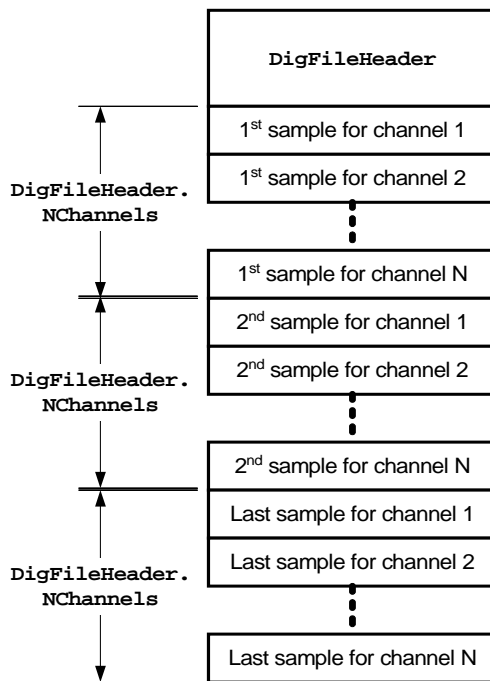
Example:
> plxread ..\SampleData\test1.plx -all
```

The main files associated with this project are listed in the table below:

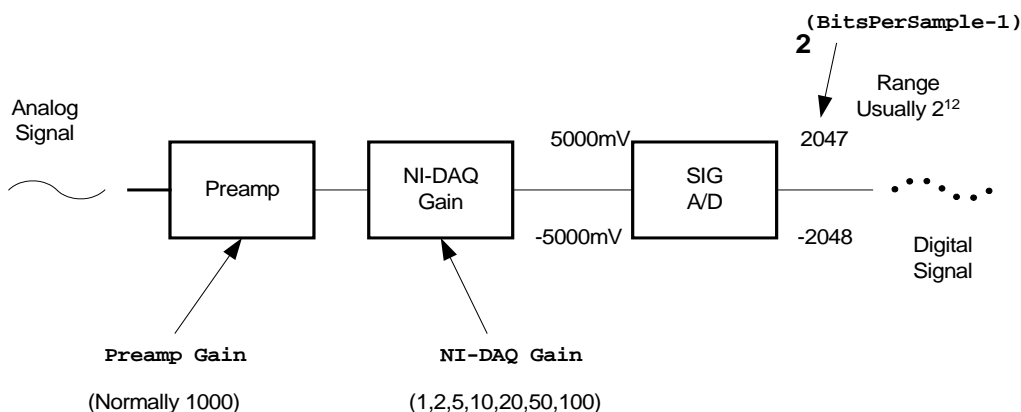
File	Description
plxread.dsp	Microsoft Visual Studio C++ 6.0 project file
plxread.cpp	Source code file for reading a PLX file.

3 DDT File Structure

A DDT file contains a file header followed by an array of A/D samples stored as 16-bit integers regardless of whether values are 12 or 16-bits.



The samples for each channel represent a digitized version of the analog signal as shown below:



The preamp is typically configured to provide a gain of 1000 and the NI-DAQ gain represents a gain of 1, 2, 5, 10, 20, 50, or 100. To convert samples of the waveform back to the original analog voltage (mV) depends on **Version** field in the header. DDT files recorded by current versions of Plexon software (as of 7/2004) have **version == 102**. Note that the total size of the header is the same for all versions.

3.1 Version 100

Version 100: Samples are assumed to be 12 bits. All channels have the same NIDAQ gain, and preamp gain is not saved.

```
struct DigFileHeader
{
    int      Version;           // =100
    int      DataOffset;        // Offset into the file where the data starts
    double   Freq;              // Digitization frequency
    int      NChannels;         // Number of channels

    int      Year;              // Time/date when the data was acquired
    int      Month;
    int      Day;
    int      Hour;
    int      Minute;
    int      Second;

    int      Gain;              // NIDAQ gain (the same gain for all channels)
    char     Comment[128];      // User-supplied comment

    unsigned char Padding[256];
};
```

The voltage in mV is determined by:

$$\text{Voltage} = \frac{(\text{sample_value})(5000)}{(2048)(\text{Gain})(1000)}$$

For this equation, Gain represents the NI-DAQ gain. The preamp gain is not specified but is assume to be 1000. Bits-per-sample is also assumed to be 12.

3.2 Version 101

Version 101: A field was added to indicate bits-per-sample (12 or 16). All channels have the same NIDAQ gain, and preamp gain is not saved.

```
struct DigFileHeader
{
    int      Version;           // =101
    int      DataOffset;        // Offset into the file where the data starts
    double   Freq;              // Digitization frequency
    int      NChannels;         // Number of channels

    int      Year;              // Time/date when the data was acquired
    int      Month;
    int      Day;
    int      Hour;
    int      Minute;
    int      Second;

    int      Gain;              // NIDAQ gain (the same gain for all channels)
    char     Comment[128];      // User-supplied comment

    unsigned char BitsPerSample; // ADC resolution, usually either 12 or 16.
    unsigned char Padding[255];
};
```


The voltage for version 101 files is determined by:

$$\text{Voltage} = \frac{(sample_value)(5000)}{\frac{1}{2}(2^{BitsPerSample})(Gain)(1000)}$$

In this equation, Gain represents the NI-DAQ gain which is the same for all channels. The preamp gain is not specified but is assumed to be 1000.

3.3 Version 102

Version 102: A byte array of per-channel NIDAQ gains was added. The Gain field now indicates the preamp gain, a value of 1 usually indicating that no preamp gain was specified in the application that recorded the DDT.

```
struct DigFileHeader
{
    int      Version;           // Version of the data format; determines which
                                // data items are valid
    int      DataOffset;       // Offset into the file where the data starts
    double    Freq;            // Digitization frequency
    int      NChannels;        // Number of channels

    int      Year;             // Time/date when the data was acquired
    int      Month;
    int      Day;
    int      Hour;
    int      Minute;
    int      Second;

    int      Gain;             // As of Version 102, this is the *preamp* gain,
                                // not ADC gain
    char      Comment[128];    // User-supplied comment

    unsigned char BitsPerSample; // ADC resolution, usually either 12 or 16.
                                // Added for ddt Version 101
    unsigned char ChannelGain[64]; // Gains for each channel. Added for ddt Version 102

    unsigned char Padding[191];
};
```

For version 102 files, the voltage is determined by:

$$\text{Voltage} = \frac{(sample_value)(5000)}{\frac{1}{2}(2^{BitsPerSample})(ChannelGain[channel])(Gain)}$$

In this equation, ChannelGain represents the NI-DAQ gain and Gain represents the preamp gain. If the Gain is set to 1, then the preamp gain was not entered by the application that created the DDT file.

3.4 Version 103

Version 103: A field was added to specify ADC maximal input voltage. The value is integer number in millivolts.

```
struct DigFileHeader
{
    int      Version;           // Version of the data format; determines which
                                // data items are valid
    int      DataOffset;       // Offset into the file where the data starts
    double   Freq;             // Digitization frequency
    int      NChannels;        // Number of channels

    int      Year;             // Time/date when the data was acquired
    int      Month;
    int      Day;
    int      Hour;
    int      Minute;
    int      Second;

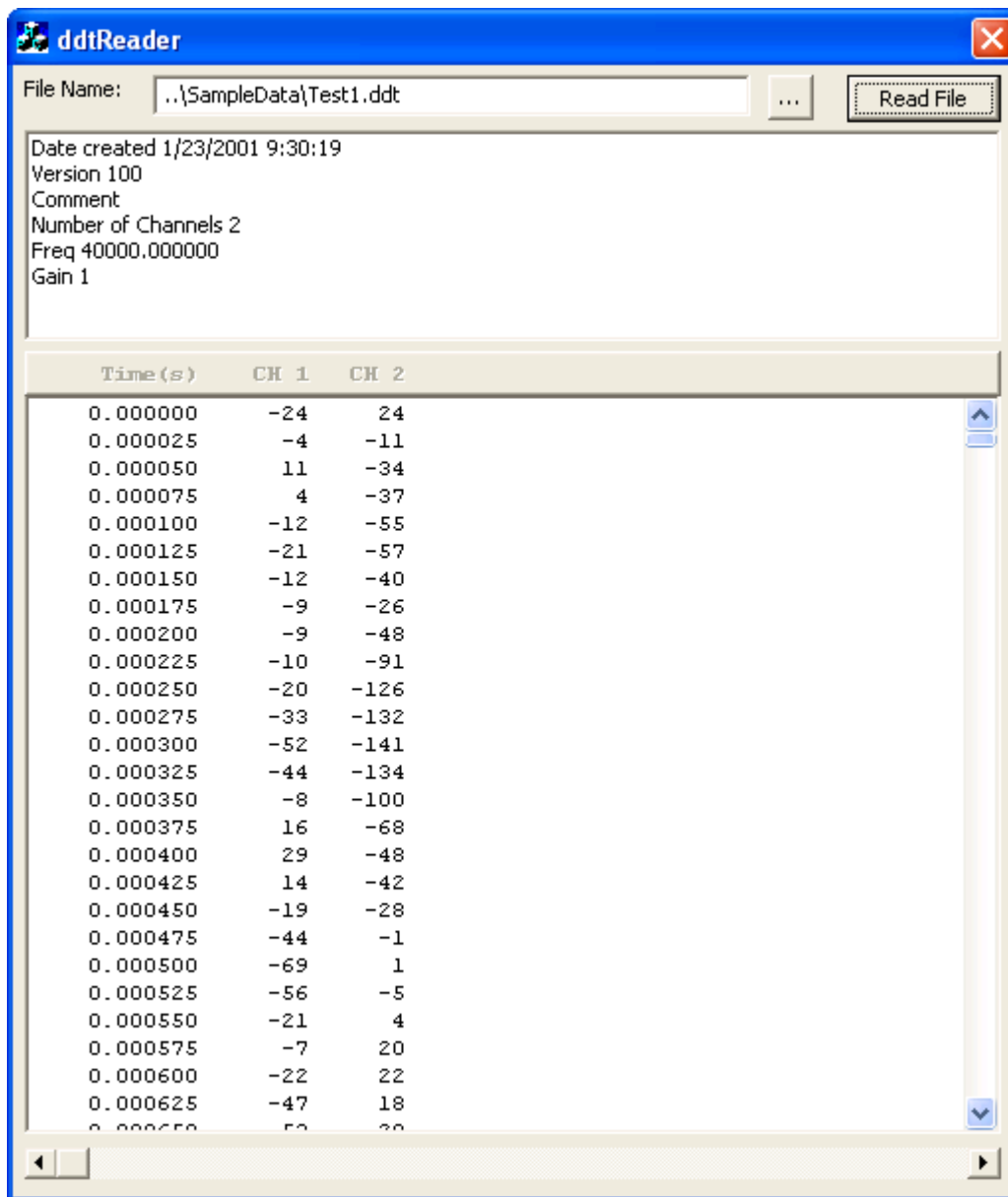
    int      Gain;             // As of Version 102, this is the *preamp* gain,
                                // not ADC gain
    char      Comment[128];    // User-supplied comment
    unsigned char BitsPerSample; // ADC resolution, usually either 12 or 16.
                                // Added for ddt Version 101
    unsigned char ChannelGain[64]; // Gains for each channel. Added for ddt Version 102
    short     MaxMagnitudeMV;    // ADC max input voltage in millivolts: 5000 for NI,
                                // 2500 for ADS64
                                // Added for ddt version 103
    unsigned char Padding[189];
};
```

4 DDT File Reader Sample Programs

Two sample applications are provided to illustrate how to read DDT files: an MFC Sample and a Win32 Console sample. Source code is provided for both samples.

4.1 MFC Sample Application

The MFC sample application is a dialog based application that displays the contents of a DDT file's header and data as shown below:

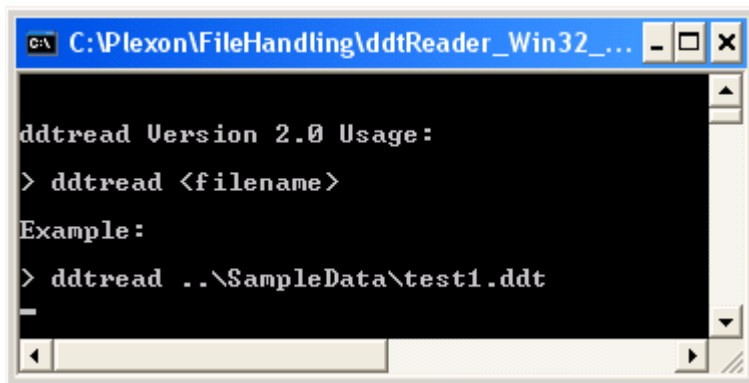


The main files associated with this project are listed in the table below:

File	Description
ddtReader.dsp	Microsoft Visual Studio C++ 6.0 project file
ddtReaderDlg.h ddtReaderDlg.cpp	Contain the header and implementation for the CDdtReaderDlg class. This class implements the main dialog for reading DDT files.
ddtReader.h ddtReader.cpp	Contains header and implementation for the CDdtReaderApp class. This class contains mostly boilerplate code for the main application.

4.2 Win32 Console Sample

TheWin32 console sample application displays the contents of a DDT file's header and data as shown below:



The main files associated with this project are listed in the table below:

File	Description
ddtread.dsp	Microsoft Visual Studio C++ 6.0 project file
ddtread.cpp	Source code file for reading a DDT file.

5 CinePlex Tracking Coordinate Decoding

Plexon .plx files may contain coordinates from the Plexon Cineplex video tracking system encoded into the strobed event words. For each video frame, there are up to 3 sets of (x,y) coordinates and potentially a motion measure encoded into up to 6 strobed events. The video tracking coordinates are identified by specific bit patterns in the strobed data words. The range of the coordinates are 0-1023 for x and 0-767 for y.

Two sample applications are provided to illustrate how to decode these coordinates from the plx files: an MFC Sample and aWin32 Console sample. Source code is provided for both samples.

Both samples use a set of re-useable routines in the files vt_interpret.h and vt_interpret.cpp.